



# CyberSec Bootcamp

## Lecture – 0x02



```
lookup.KeyValue  
f.constant(['em  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```

# MEET YOUR GUIDES :



**Zishan Ansari**  
(Mentor)  
**CyberSec - GDG**



```
lookup.KeyValue  
f.constant(['en  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```


# MEET YOUR GUIDES :



**Rohit Choudhary**  
(Instructor)  
CyberSec - GDG



```
lookup.KeyValue  
f.constant(['em  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```



```
lookup.KeyValue
tf.constant(['em
=tf.constant([0
lookup.StaticV
buckets=5)
```



# TODAY'S OBJECTIVES-

- ❖ **Controlling File And Directory Permissions**
- ❖ **Process Management**
- ❖ **Managing User Environment Variables**



```
lookup.KeyValue  
f.constant(['em  
=tf.constant([G  
.lookup.StaticV  
_buckets=5)
```



# Controlling File and Directory Permissions -

\$> Not every user of a single operating system should have the same level of access to files and directories. Like any professional or enterprise-level operating system, Linux has methods for securing file and directory access. This security system allows the system administrator-the root user - or the file owner to protect their files from unwanted access or tampering by granting select users permissions to read, write, or execute files.

\$> In Linux, r refers to read, w refers to write, and x refers to execute

\$> chown "username" "file" : We use this command to give ownership of a particular File to a particular user

```
lookup.KeyValue  
f.constant(['en  
=tf.constant([G  
.lookup.StaticV  
_buckets=5)
```



# Controlling File and Directory Permissions -

**\$> chgrp “group name” “file”** : We use this command to give ownership of a particular file to a group

**\$> ls -l “file”** : With the use of this command, we can get the whole list of information about the file along with which user have what permission to that file

```
kali>ls -l /usr/share/hashcat
total 32952
❶❷❸❹❺❻❼
drwxr-xr-x 5 root root 4096 Dec 5 10:47 charsets
-rw-r--r-- 1 root root 33685504 June 28 2018 hashcat.hcstat
-rw-r--r-- 1 root root 33685504 June 28 2018 hashcat.hctune
drwxr-xr-x 2 root root 4096 Dec 5 10:47 masks
drwxr-xr-x 2 root root 4096 Dec 5 10:47 OpenCL
drwxr-xr-x 3 root root 4096 Dec 5 10:47 rules
```

- ❶ The file type (this is the first character listed)
- ❷ The permissions on the file for owner, groups, and users, respectively (this is the rest of this section)
- ❸ The number of links
- ❹ The owner of the file
- ❺ The size of the file in bytes
- ❻ When the file was created or last modified
- ❼ The name of the file

```
lookup.KeyValue
f.constant(['em
=tf.constant([G
lookup.StaticV
_buckets=5)
```

# Controlling File and Directory Permissions -

**\$> Changing Permissions with Decimal Notation :** We can use a shortcut to refer to Permissions by using a single number to represent one rwx set of permissions. Permissions are represented in binary so ON/OFF switches are represented by 1 and 0, You can think of rwx permission as three ON/OFF switches, so when all permissions are granted this equated to 111 in binary. A binary set like this is easily represented as one digit by converting it into Octal, an eight-digit number system starting with 0 and ends with 7.

**Table 5-1:** Octal and Binary Representations of Permissions

Binary	Octal	rwx
000	0	---
001	1	--X
010	2	-W-
011	3	-WX
100	4	Y--
101	5	Y-X
110	6	YW-
111	7	YWX

So, when we will give a user a permission to only read we will write 4 as octal and when will give full permission (all switches are on) they are represented as octal equivalent of 7.

Refer next slide for examples

```
lookup.KeyValue  
f.constant(['em  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```



# Controlling File and Directory Permissions -

Using this information, let's go through some examples. First, if we want to set only the read permission, we could consult Table 5-1 and locate the value for read:

---

```
r w x
4 - -
```

---

Next, if we want to set the permission to `wx`, we could use the same methodology and look for what sets the `w` and what sets the `x`:

---

```
r w x
- 2 1
```

---

Notice in Table 5-1 that the octal representation for `-wx` is 3, which not so coincidentally happens to be the same value we get when we add the two values for setting `w` and `x` individually:  $2 + 1 = 3$ .

Finally, when all three permissions are on, it looks like this:

---

```
r w x
4 2 1
```

---

And  $4 + 2 + 1 = 7$ . Here, we see that in Linux, when all the permission switches are on, they are represented by the octal equivalent of 7.

**Fig. 3**

So, if we wanted to represent all permissions for the owner, group, and all users, we could write it as follows:

---

```
7 7 7
```

---

**Fig. 4**

---

```
kali >ls -l
total 32952
drwxr-xr-x 5 root root 4096 Dec 5 10:47 charsets
❶ -rwxrwxr-- 1 root root 33685504 June 28 2018 hashcat.hcstat
-rw-r--r-- 1 root root 33685504 June 28 2018 hashcat.hctune
drwxr-xr-x 2 root root 4096 Dec 5 10:47 masks
drwxr-xr-x 2 root root 4096 Dec 5 10:47 OpenCL
drwxr-xr-x 3 root root 4096 Dec 5 10:47 rules
```

---

You should see `-rwxrwxr--` on the left side of the `hashcat.hcstat` line ❶. This confirms that the `chmod` call successfully changed permissions on the file to give both the owner and the group the ability to execute the file.

**Fig. 5**



# Controlling File and Directory Permissions -

**\$> Changing Permissions with UGO :** The symbolic method to change permission is referred as UGO syntax (UGO stands for User/Owner, Group, and Others). UGO syntax is very simple. Enter the `chmod` command and then the users we want to change permissions for, `u` for user, `g` for group and `o` for others.

**\$> The syntax for UGO method is:**

- (minus) : Removes a permission
- + (plus) : Adds a permission
- = (equal) : Sets a permission

After the operator, include the permission you want to add or remove (`rw`) and the name of the file. Eg: `chmod u-w "file"` : This command says to Remove (-) the write (`w`) permission from "file" for user (`u`)



# Controlling File and Directory Permissions -

**\$> chmod stands for change mod**

**\$> chmod +t “file name”** : Give permission of sticky bit, The sticky bit is a permission bit that can be set on directories/files to control how files within those directories are managed. with sticky bit only the file's owner/directory owner or the root user can delete, or rename files within that directory.

**\$> SUID:** Stands for Set Owner User ID up on execution is a file permission in Linux. It gives a user temporary permission to run a file or program with the same permissions as the file owner. Use the command “chmod u+s file” or Add 4 to the beginning of the octal permissions, for Eg – “chmod 4755 file”

```
lookup.KeyValue  
f.constant(['en  
=tf.constant([  
lookup.StaticV  
_buckets=5)
```



# Process Management -

\$> At any given time, a Linux system may typically has hundreds, or sometimes even thousands, of processes running simultaneously. A process is simply a program that's running and using resources. Eg – Terminal, Web Server, Any running commands, Databases

\$> ps : To view processes (just like how task manager is for windows)

\$> ps aux : Will show all processes running on the system for all users

```
kali >ps aux
USER  PID  %CPU  %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root    1   0.0   0.4 202540 6396 ?        Ss   Apr24   0:46 /sbin/init
root    2   0.0   0.0      0     0 ?        S    Apr24   0:00 [kthreadd]
root    3   0.0   0.0      0     0 ?        S    Apr24   0:26 [ksoftirqd/0]
--snip--
root  39706  0.0   0.2  36096  3204 pts/0    R+   15:05   0:00   ps aux
```

**USER** The user who invoked the process

**PID** The process ID

**%CPU** The percent of CPU this process is using

**%MEM** The percent of memory this process is using

**COMMAND** The name of the command that started the process

```
lookup.KeyValue
f.constant(['en
=tf.constant([G
lookup.StaticV
_buckets=5)
```



# Process Management -

**\$> sleep “time(suffix)”** : This command stops the whole terminal for the given amount of time (default is seconds) we have mentions. For Eg – “sleep 100” will stop the terminal for 100 seconds, to cancel the sleep we can wait for 100 seconds or just simply do ctrl+z to suspend this

**\$> You can temporarily stop a running process using ctrl+z**

**\$> bg** : This command resumes the stopped process and continues running it in background

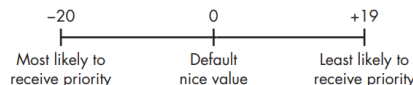
```
lookup.KeyValue  
f.constant(['en  
=tf.constant([G  
.lookup.StaticV  
_buckets=5)
```

# Process Management -

**\$> ps aux | grep "Process name" : To check if a particular process is running or not**

**\$> top : This command display the greediest processes (Processes with most resources)**

**\$> Changing Priority with "nice" : The nice command is used to influence the priority of a Process to the kernel. The value of nice range from -20 to +19, with zero (0) being default value. A high nice value translates to a low priority, and a low nice value translates to a high priority. The owner of the process can lower the priority of the process but cannot increase it's priority. Of course superuser (sudo/root) can set priority to whatever they like.**



```
lookup.KeyValue  
f.constant(['en  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```

# Process Management -

\$> nice -n "value" "process" : Set the priority which we will enter in the "value". Eg: "nice -n -10 /bin/process" will increment the nice value by -10, increasing its priority and allocating it more resources

\$> Changing the Priority of a Running Process with renice : The renice command will take absolute values between -20 and 19 and sets the priority to that particular level, rather than increasing or decreasing from the level at which it started

\$> renice "value" "process" : Will assign absolute "value" to the process

```
lookup.KeyValue  
f.constant(['en  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```



# Process Management -

**\$> Killing Processes :** At times, a process will consume way too many system resources, exhibit unusual behavior, or – at worst – freeze. A process that exhibits this type of behavior is known as “rogue process”. So we kill such type of processes.

**\$> The kill command has 64 different kill signals, and each does some-thing slightly different . The syntax for the kill command is “kill –signal PID”, PID refers to Process ID and the signal switch is optional. By default signal flag is set as SIGTERM.**

**\$> Refer next slide for most common Kill signals.**

```
lookup.KeyValue  
f.constant(['en  
=tf.constant([G  
.lookup.StaticV  
_buckets=5)
```



# Process Management -

Signal name	Number for option	Description
SIGHUP	1	This is known as the <i>Hangup (HUP)</i> signal. It stops the designated process and restarts it with the same PID.
SIGINT	2	This is the <i>Interrupt (INT)</i> signal. It is a weak kill signal that isn't guaranteed to work, but it works in most cases.
SIGQUIT	3	This is known as the <i>core dump</i> . It terminates the process and saves the process information in memory, and then it saves this information in the current working directory to a file named <i>core</i> . (The reasons for doing this are beyond the scope of this book.)
SIGTERM	15	This is the <i>Termination (TERM)</i> signal. It is the kill command's default kill signal.
SIGKILL	9	This is the absolute kill signal. It forces the process to stop by sending the process's resources to a special device, <i>/dev/null</i> .

\$> If you just want to restart a process with HUP signal, enter the -1 option with kill “kill -1 PID” or if we want to go for an absolute kill of a rouge process we will using the signal -9 with the kill command “kill -9 PID”

\$> If we don't know the PID, we can use “killall” command to kill the process.

```
lookup.KeyValue  
f.constant(['em  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```



# Process Management -

**\$> fg PID :** To move a process running in the background to foreground (fg)

**\$> Scheduling Processes:** Both Linux system administrators and hackers often need to Schedule processes to run at a particular time of day. For this we will use “at” Command

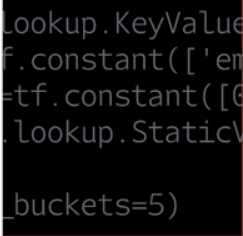
Time format	Meaning
at 7:20pm	Scheduled to run at 7:20 PM on the current day
at 7:20pm June 25	Scheduled to run at 7:20 PM on June 25
at noon	Scheduled to run at noon on the current day
at noon June 25	Scheduled to run at noon on June 25
at tomorrow	Scheduled to run tomorrow
at now + 20 minutes	Scheduled to run in 20 minutes from the current time
at now + 10 hours	Scheduled to run in 10 hours from the current time
at now + 5 days	Scheduled to run in five days from the current date
at now + 3 weeks	Scheduled to run in three weeks from the current date
at 7:20pm 06/25/2019	Scheduled to run at 7:20 PM on June 25, 2019

Time Formats Accepted by the “at” command

**\$> When we use “at” command with the specified time, “at” goes into interactive mode and we are greeted with an interactive mode**

```
kali >at 7:20am
at >/root/myscanningscript
```

This code snippet will schedule “myscanningscript” to execute today at 7:20 am



# Managing User Environment Variables -

\$> To get the most from your Linux hacking system, you need to understand environment variables and be adept at managing them for optimal performance, convenience, and even stealth.

\$> Technically, there are 2 types of variables:

1. **Environment Variables:** Are process-wide variables built into your system and interface that control the way your system looks, acts, and “feels” to the user, and they are inherited by any child shells or processes

2. **Shell Variables:** Are typically listed in lowercase and are only valid in the shell they are set in

```
lookup.KeyValue  
f.constant(['en  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```



# Managing User Environment Variables -

**\$> env :** Use this command to view all default environment variables, Environment variables are always in uppercase, as in HOME, PATH, SHELL, and so on

**\$> set | more :** To view all environment variables, including shell variables, local variables, and shell functions such as any user-defined variables and command aliases.

**\$> set | grep "variable name" :** To grep/filter out a particular variable

**\$> "variable name"="variable" :** To change the value of a variable for a session

```
lookup.KeyValue  
f.constant(['em  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```



# Managing User Environment Variables -

**\$> Making Variable Value Changes Permanent :** When you change an environment variable, that change only occurs in particular environment . This means that when you close the terminal, any changes you made are lost, with values set back to their defaults. To make the values permanent we will use “export” command. This command will export the new value from your current environment to any new forked child processes.

**\$> Firstly set the value of variable “variable name”=“variable” and then use “export”**  
**By writing: export “variable name” as shown below**

---

```
kali > HISTSIZE=1000  
kali > export HISTSIZE
```

---

This code snippet will set your HISTSIZE variable's value to 1,000 and export it to all your environments.

# Managing User Environment Variables -

**\$> Changing Your Shell Prompt :** Your shell prompt, another environment variable, provides you with useful information such as the user you're operating as and the directory in which you're currently working. You can change the name in the default shell prompt by setting value for "PS1" variable. The PS1 variable has a set of placeholders for information you want to display in the prompt, \u for The name of current user, \h for The hostname, and \w for The base name of the working directory.

---

```
kali >PS1="World's Best Hacker: #"  
World's Best Hacker: #
```

---

**\$> Use "export PS1" to save the changes permanently**

**\$> echo \$PATH :** Tell the location of path variable



# Managing User Environment Variables -

**\$> Path variable is one of the most important variables in our environment, this controls where on your system your shell will look for commands you enter such as grep, ls, and echo. Most commands are located in the sbin or bin subdirectory, like /usr/local/bin(or sbin)**

**\$> Adding to the PATH Variable :** If you downloaded and installed a new tool-let's say newhackingtool-into the /root/newhackingtool directory, we can only use commands from this tool only when we will be in the same directory as the tool, so to add this directory in path variable we will use the command "PATH=\$PATH:/root/newhackingtool"

**\$> Creating a User-Defined Variable:** We can create a user-defined variable by simply assigning a value to a variable

```
kali > MYNEWVARIABLE="Hacking is the most valuable skill set in the 21st century"
```

```
lookup.KeyValue  
f.constant(['en  
=tf.constant([G  
lookup.StaticV  
_buckets=5)
```



# THANK YOU



```
lookup.KeyValue  
f.constant(['em  
=tf.constant([0  
lookup.StaticV  
_buckets=5)
```